計算機屋から見た計算の複雑さ

竹内 郁雄 (NTT ソフトウェア研究所)

計算機は複雑さの宿命を背負って生まれてきた工学的所産である．その歴史は複雑さの回避の歴史であったともいえる．しかし，これからは逆に複雑性を楽しむというジャンルも生ずるであろう．最後に，複雑性とはそもそもなにかについて若干の考察をする．

1. はじめに

どんなものにも定義を与えなければならないと気がすまないというほど頑固でないにしても，定義がないと落ち着かないのは，われわれ技術屋の性である．反面，だからこそ面白いと感じるのも，研究者の端くれの性である．

計算機科学における計算量的複雑度や Kolmogorov 複雑度以外に，「複雑」という言葉の厳密な定義を与えようとした試みを知らない筆者には，「複雑」という言葉がとびかう昨今の「複雑系ブーム」に多少「複雑」な感想がある．しかしいま，複雑なものが面白いこともたしかである．

計算機は生まれたときからどころか，その誕生以前から深刻な複雑性の問題をわれわれにぶつけてきた．すべからく工学は，多かれ少なかれ複雑性と闘ってきた歴史をもつ．計算機の場合は，複雑性との闘いが，たとえば「ソフトウェア危機」という言葉で表現されるレベルにまで至った．

計算機はおそらく人間の工学的所産としては最も複雑なシステムである．ここでとりあえず，「複雑」とは人間にとってわかりにくいというナイーブな意味である．人間にとってわかりにくければ，当然取りあつかいにくい．そんなものは工学的所産としては許されないのである．本稿では，まず，このナイーブな意味での複雑さと闘ってきた計算機科学・工学の歴史についてざっと振り返る．

複雑性との闘いはこれからも工学の本道だろうが，筆者は最近，このような複雑さを享受して楽しむことに，これからの計算機の一つの活路があるように思えるようになってきた．これがテクノロジの枠組みを超えた次世代の計算機のあり様を示唆しているように思えてならないのだ．これについても 2, 3 のキーワードを挙げよう．

最後に，やはり「複雑さ」とはなにかについて考えるための所感を述べる．これが，計算量的複雑性などで定義される「複雑性」と，われわれが直観として感ずる「複雑性」の関係を考えるためのヒントになればさいわいである．

2. 複雑さと闘ってきた計算機

最近のパソコンのふたをあけると，整然とならんだ黒いキャラメルのようなものが見えるだけである．はるかに単純な論理構造をもつ黎明期の計算機のほうが，おどろおどろしい配線のトグロで見る者を圧倒する．もっとも，配線のトグロのものすごさは計算機の黎明期だけではなくて，つい 10 年前でも，研究室レベルの実験機ではよく見られた．

筆者の知る最善 (?) の例は，後藤英一東大教授 (当時) らが理研で開発した FLATS という ECL (Emitter Coupled Logic) マシンである．ECL は集積度が低いため，ECL の中規模集積回路 (MSI) が載った多数の基板をバックプレーンに差し，バックプレーン上に並んだ 1000 本以上のピンの間を，長さを揃えたツイストペアのラッピングで配線する．こういうふうにして組み立てられた大型冷蔵庫大の筐体が全部で 5+2 個，中心に五角形の空間が空くように配置されていた (筐体間の配線長を最小にする合理的な配置である)．最も重要な CPU 筐体のバックプレーンのラッピングはなんとワイアの束が厚さ 25 センチほどになっていた — つまり，大盛ソバのようになっていた．ラッピングピンは見えるどころか，線をかきわけることすら困難で，デバグのための配線のやり直しは特殊技能者にのみ可能だったという．筆者の研究所でも 10 年ほど前，フラットケーブルが山になってつながったキシメンザルのような並列計算機の研究が行なわれたことがある．

研究室ではいざ知らず，大量生産と保守性・信頼性を旨とする近代工業製品である計算機にとって，このような複雑さを回避することは至上命題であった．その成果はハードウェア技術者が依ってたつ中心的スキルの変遷に見ることができる．フリップフロップ回路のアナログ技術から，AND や OR 回路をバックした MSI の利用

技術，そして各種専用LSIを組み合わせてつかう技術と，あつかう部品は物理的には小型化したが，論理的には多機能化してきた．部品の論理的多機能化は，部品自身の複雑性を増したが，さいわいなことに内部の複雑さをブラックボックスの内側に隠蔽することに成功したのである．

複雑さをブラックボックスという部品化によって，視界から追いやる．これがどういう場合でも成功するとはかぎらないことに注意しておこう．成功するのは，一般に，われわれの「理解のレベル」で，ブラックボックスの機能を語ることができる場合のみである．このときは，「それ以下のレベルの内部的複雑さ」はシステムの当面の理解には不要なものとなる．

ハードウェア技術における複雑さの回避は，部品の論理的多機能化をブラックボックス化することによって成功し続けてきた．しかしソフトウェアでは，このような部品化がハードウェアほどには成功しなかった．というより，ほとんど成功しなかった．たとえ，ソフトウェアの個々の技術要素としての成功がいくばくかあったとしても，ソフトウェアに要求される機能の複雑さの伸びにまったく追いつかなかった．

同じプログラムステップをパラメータを変えて何回もつかうサブルーティンは，ソフトウェアにおけるおそらく最初の部品概念である．プログラミング言語が機械語から，より記述性の高い「高級」言語に変遷するにつれ，サブルーティンは手続きや関数の概念へ進展していったが，ソフトウェアの複雑さに対処する有効打にはならなかった — とはいっても，これらがなければソフトウェアなど端からつくりようがなかったことも事実である．

一見，ハードウェア技術における部品化，ブラックボックス化に相当するこれらの概念があまり有効でなかった理由は，まとめると次の三つになるだろう．

(1) ブラックボックス化，すなわち内部情報の複雑さの隠蔽に成功しないことが多かった．つまり，外側に見える仕様と，内側での実装の切り分けがうまくいかず，外側にまで，内部事情が見えてしまうことが多かった．これでは，ブラックボックスでなくて，まだらボックスである．部品としての仕様が複雑になってしまい，部品として再利用される機会も減ってしまう．ハードウェアの場合は，技術の階層分けが最初から確立していたのに，ソフトウェアでは，階層分けが便宜的でしかあり得なかったのだ — 記号論理はどこで切っても記号論理で，まるで金太郎飴である．

(2) ソフトウェアのもつ本質的な記述能力の高さ (書くだけならなんでも書ける!?) と，ソフトウェアに要求された機能の多様さが，このような部品化がもたらす概念整理のレベルをはるかに超えていた．部品化はある意味で焼石に水であった．逆に部品化をまともにやろうとすると，手に負えないほどの大量の部品種類が必要になり，部品化自身が新たな複雑性の問題を生み出すことになってしまう．

(3) 上と双対の理由だが，部品の仕様の記述自体が複雑だった．仕様の厳密な記述が，内部実装のための記述 (プログラム) と同程度の複雑さ (この場合は記述に要する文字の量と考えてよい) をもってしまうという，冗談みたいなことが現実に起こる．これは人間の理解のレベルとの整合性をとるという部品化の意義と矛盾してしまう．

これらの理由のさらに背後には，人間とソフトウェアの関わりに関するもっと根源的なものが潜んでいる．筆者の独断によれば，それは人間の空間認知能力と時間認知能力の差である．以下，この視点に沿って歴史を追ってみよう．

当初，ハードウェアは，一度つくったら変更不能の固い技術で，コストのかかる困難な技術と考えられていた．それに対して，ソフトウェアはいつでも変更可能なやわらかい技術で，保守性もよく，コストの低い技術と考えられていた．1960年代，計算機システムは急速な成長をとげ，科学技術計算，事務計算の両分野で広くつかわれるようになったが，ソフトはハードを買ったときにオマケでついてくるものであった．ところが，ソフトウェアがハードウェアよりもずっとあつかいにくい質の悪い技術であることに計算機屋が気づくのにそう時間はかからなかった．60年代に流行語となった「ソフトウェア危機」は本来，ソフトウェアの需要に供給能力が追いつかず，ソフトウェアの不足が重大な社会問題になるという危機感を表わす言葉であるが，これはそのままソフトウェアの難しさを言い表していた．

1960年代の記念碑的論文であるDijkstraの "Goto considered harmful" (goto有害論, 1968) はわずか2ページのレターであるが，プログラムの中でのgoto文がいかにプログラムの理解を困難にするかについて鋭い指摘をし

た．プログラムを読んでいるときにgotoが出てくると，その行き先を探し，そこへ視点を移さなければならない．こういった視点の移動は，書かれているプログラムテキストの空間的構造と無関係であり，読む人間にあてどもない彷徨を強要する．こういう彷徨の中からプログラムが実現しているアルゴリズムの抽象的構造を再構築して理解することは，非常に困難である．当然，プログラムの修正・改造も，修正の影響範囲がすぐにはわからないので困難である．

この簡明直截にして的を得た指摘は，1970年代の「構造化プログラミング」ブームをもたらした．構造化プログラミングとは，プログラムの論理的抽象構造と，プログラムテキストの空間構造をなるべく一致させようとする考え方である．低レベルでは，プログラムテキストを，逐次実行，選択 (if-then-else)，繰り返しの三つの要素の結合で表現し，gotoのように実行順序とプログラムテキストの構造の不一致を起こすものをつかわないようにする．もっと高いレベルでは，大規模なプログラム全体を分割統治 (divide and conquer) によって段階的に分割し，各段階で人間の理解のレベルに合わせた詳細化を行なう．後者は，システム工学におけるトップダウン手法，まさに還元的手法そのものである．

プログラムテキストの構造に統制を加える考え方は，それに適した新しいプログラム言語を多数生み出し，またプログラミング書法全般にも大きな影響を与えた．しかし，ソフトウェアのもつ本性的複雑さは，構造化プログラミングの網をもってしても押えきれるものでなかった．プログラムはますます巨大化し，複雑化していったのである．ちなみに，最近あるソフトウェア業界の人から聞いた話では，銀行のATMシステムのプログラム300万行(!)中，利息計算のためなどのために預け入れ日数を計算するモジュールのプログラム行数がなんと10万行だという (コメント行ではない)．通常のカレンダー計算のプログラムが高々数行で書けるという事実を知れば，この恐るべき巨大さの摩訶不思議さが実感されるであろう．巨大さイコール複雑さとはかぎらないが，一般にトップダウンでつくられた大規模ソフトがつねに予想よりも巨大なステップ数になるのは，世の中では常識である．

構造化プログラミングにおいても，プログラムの適切な抽象化の必要性が主張されていたが，1980年代にブームとなったオブジェクト指向プログラミング (あるいはもっと一般的にオブジェクト指向概念) によって，プログラム抽象化の意義や意味がより明確になった．それまでのプログラミング (オブジェクト指向に対して，手続き指向と呼ぶ) では，プログラムステップをまとめあげて，手続きや関数，さらにプログラムモジュールと，抽象化の段階を積み上げていったのに対し，オブジェクト指向では，データ構造を基礎として抽象化を積み上げていく．すなわち，手続き指向は，実行手順という，いわば時間的な概念を基礎にした抽象化を行なったのに対し，オブジェクト指向では問題を記述するときに出てくる「もの (object)」，たとえば画面上のウィンドウや，運送伝票，倉庫の一室といったものを計算機内部にモデル化した「もの」の概念にもとづいて抽象化を進める．これは人間が，見えるものから出発して抽象概念を獲得していく認知プロセスとも相性がよい．

オブジェクト指向プログラミングにおける「もの」すなわちオブジェクトは，対象をモデル化したデータ構造と，それをあつかう手続きを一体化したものである．オブジェクトと外界のインタラクションは一般にメッセージと呼ばれる簡略な通信プロトコルだけを介して行なわれる．手続き指向にくらべて，データ構造の内部構造を知らずにすむだけ，いろいろな話が簡単になる．

オブジェクトはブラックボックス化が一段と進んだものであったため，ソフトウェア部品としての価値がクローズアップされることになった．メッセージプロトコルという簡潔な仕様により，部品の再利用性が高まると考えられたのである．しかし，これも当初期待されたほどうまくはいかなかった．これは現在でも研究課題のまま積み残されているといってよいだろう．やはりソフトウェアの複雑さは一筋縄では対処できないのである．

それでは60～70年代のソフトウェア危機はいまだに続行したままかというと，以前ほどの危機感がなくなったことも事実である．遅々としているものの，ソフトウェアの要素技術は着実に進歩しているうえ，つくるべきプログラムの種類がそう青天井でもなくなってきたからであろう．それに，人々がソフトウェアに「慣れてきた」ということも大きい．1960年当時はFortranコンパイラの開発は30人年とされていた．しかし現在，よくできる学部学生であれば，半年でそれなりのコンパイラをつくれるようになっている．その差を裏付けるソフト

ウェアの進歩はなんだったのか?

筆者は，どこにもエポックはなかったと思う．しかし，ソフトウェアという，人間の言葉に近い言葉で書かれるものには，人間のつくってきた言葉の文化と同じような，まさに文字どおりの文化的蓄積があって，結局それが，生産性の差を生んだのであろう．それは，有用な概念を短く言い表す語彙の発明と蓄積であり，教育の改良であり，ソフトウェアにかかわる人口あるいはコミュニティの拡大である．一言でいうなら，まさに人々がソフトウェアに慣れてきて，理解のレベルが変わってきたのである．

こういうソフトウェア文化の蓄積は，ソフトウェアのある種の固定化でもあった．以前はあちこちでプログラミング言語やオペレーティングシステムの設計開発が行なわれていたが，いまやその数は激減した．悪くいえば，言語やOSの寡占化が進んだのだが，文化の醸成とは所詮そんなものだという見方も成り立つ．(もっとわかりやすい例ではワープロソフトがある．一時あれほど群雄割拠であったこの世界もいまや寡占化が完了してしまった．困るのは，それが技術の頂点をきわめるかなり前に終わってしまったことである.) このような固定化が，ソフトウェア全体としての複雑性を減ずることはあきらかだろう．いずれにせよ，純技術的現象というより社会現象として，ソフトウェアの問題が変遷していきそうな気配となった．余談だが，このように固定化するソフトウェアに対して，いまやハードウェアのほうがやわらかく対処していこうという傾向があるのは興味深い．

こういう大きな潮流は，野放図にソフトウェアをつくることによって直面せざるを得なかった複雑さの問題を，社会全体で (多分，経済原則にもとづいて) 対応し，回避していこうという人間社会のもつ適応性の一つの現われだったといえよう．

少し話がそれてしまったが，構造化プログラミング，オブジェクト指向プログラミングという歴史の本当の脊骨のようなストリームをたどってみると，プログラミングを人間にとってやさしくする (複雑でなくする) ための技術の根底には，プログラミングを時間的認知の対象から空間的認知の対象へ移そうという運動がある．構造化プログラミングは，プログラムテキストの空間構造によってプログラムの抽象構造を表現しようとした．また，オブジェクト指向プログラミングは，手順という時間構造ではなく，ものという空間構造を中心に据えたプログラム抽象へと，コペルニクス的転回を行なった．別の言葉でいえば，これらはプログラムの抽象構造の可視化にほかならない．ここでは触れないが，プログラムを文字どおり図式表現するような方法論も，このような可視化の努力の一種である．

なぜ，空間表現したほうがわかりやすいかについて，筆者にここでちゃんと述べるほどの能力もスペースもない．だが，時間解像度が高く，順序記憶のよい聴覚をベースにした時間認知にくらべて，一目でいろいろなものの関係を把握することができる視覚をベースにした空間認知のほうが，より複雑なものの理解に適していることは容易に想像できる．ハードウェア技術の進歩が，ソフトウェア技術の進歩にくらべて，微分係数であきらかに優っているのは，ハードウェア (たとえば，論理回路) が本質的に視覚的対象だからであろう．

プログラムの実行は時間的現象であるから，これを空間構造に (筆者の好きな比喩でいえば，フーリエ変換して) 焼き直す作業が必要で，また逆にそのような空間表現を時系列として実行可能にする翻訳技術も必要である．プログラミングを人間にとってやさしくするための技術的な流れは今後も可視化という線に沿っていくと筆者は考えている．実際，並列プログラミングでは時系列のうまい捨象，あるいは可視化なくしては，なにもできなくなってしまうだろう．

3. 複雑性を楽しむ

これから計算機は，実用道具の枠組みを超えて，ありとあらゆる領域で人間とかかわっていく．ここでは，日々計算機と格闘あるいは戯れている筆者が，前節で述べた，複雑性の回避とはまったく裏腹に，複雑性を楽しむという新たな境地を見出したことについて簡単に述べておきたい．

筆者は共同研究者らとともに現在，SILENTというマシンとその上のTAOという専用言語の研究開発を行なっている (これはいまどきとしては珍しい，口の悪い人にいわせれば石器時代のような研究である)．SILENTは80ビットの水平型マイクロプログラムで制御されるマシンである．VLSI CPUチップも動きだし，筆者はマイクロプログラミングとそのデバグに余念のない日常を送っている．

水平型マイクロプログラムといっても，ふつうの機械

語よりちょっと複雑かなという程度のものであるが，基本的にすべての命令に next address，すなわち，次に実行すべき番地の情報が入る．要するに goto だらけなのである．また，条件分岐が非常に多くて，数ステップ連続してすべて条件分岐というケースも少なくない．もちろん，プログラムテキスト上では，上から順に読み，大体その順序で実行されるようになっているのだが，それでも一般的につかわれる，たとえば C 言語などにくらべてはるかに低レベルである．当然，プログラミングもデバグもかなり難しい．

しかし，だからこそ，マイクロプログラムは楽しいと，最近確信をもって言えるようになった．プログラミングが大好きという人間は昔からたくさんいたし，いまでもたくさんいると思うのだが，これを単なる変わり者とか，プログラミングフリークといって片づけてはいけないという気がするのである．プログラミングを楽しむというジャンルが確立されるべきなのだ．計算機にはそれだけの広さがある．

なぜマイクロプログラムが楽しいか．それは複雑だからである．その複雑さの由来は，1 命令の中でいろいろな動作が並列に行なえるという水平型マイクロプログラムのもつややこしさもあるが，Dijkstra が喝破した有害な goto に満ちており，プログラムテキストの構造がプログラムの抽象構造をまったく反映していないことに起因する．さらに筆者らに特有の事情が加わる．SILENT が自作の LSI であるため，ところどころ仕様通りには動かないハードバグがある上，ある命令と別のある命令を組み合わせたときにだけちゃんと動かないという複合バグが出る．つまり，ハードが 100 パーセントは信用できないという切れかかった綱の上の綱渡りのようなスリルもあるのである．

筆者は松岡正剛氏に「物語」というものの面白さについて教えられたのだが，マイクロプログラムを追う楽しさはまさに物語を追う楽しさなのである．物語と同様，プログラムは時系列を追うように読まなければならない．時系列をなぞれば，なぞった道筋で起こった「事件」を記憶しなければならない．この記憶は生のままではなく，ある種の記憶術に従った構造化記憶である．普遍的物語構造には，この種のステレオタイプがいくつかあると言われているのだが，プログラムにもとっても事情は同じだと思う．無限のバリエーションなどあり得ないのだ．こうして読み解いた物語の解釈が，計算機に無残にも拒絶されると，デバグの開始である．ここから先は，推理小説の読み方になる．道筋で起こったどんな些細なことも，物語解釈のずれの解決の糸口になり得る．

バグは，おおまかにいって，アルゴリズムのバグ，アルゴリズムを正しくプログラムに翻訳していなかったプログラミングバグ，レジスタ番号などをタイプミスしたレベルのバグ，そしてハードバグがある．どれも本人が意図して仕組んだものでないので，デバグにはつねに発見の喜びがある (ハードバグの場合は，直しようがないので多くの場合悲しみとなるが…)．研究として行なっている作業の一部を「楽しみ」と称するのは不遜といわれるかもしれないが，辛い作業だからこそ見方の転換が必要なのである．

プログラムを物語として見る．これを敷衍すると，約 2 年前の情報処理学会シンポジウムで筆者が提唱した「物語プログラミング」という概念に至る．物語とプログラムは似たもの同士だというのが基本的な発想である．詳細はここで述べないが，プログラミングのもつ複雑さを逆手にとろうというわけである．「物語プログラミング」はまだ問題提起の段階であるが，物語性の強いゲームソフト (アドベンチャゲーム，ロールプレイングゲーム，シミュレーションゲーム) をどう面白くつくるかという実利的な価値をもつようになるかもしれないし，プログラムを新しい物語のジャンルに入れるきっかけになるかもしれない．ここでのキーワードは「複雑さを楽しむ」ことであり，「楽しめる複雑さ」の追求である．

このようなゲームでは，作者が，ユーザとゲームのかかわり方をどう制御するかが問題になる．たとえば，アドベンチャゲームでは作者の制御が直接的であり，シミュレーションゲームでは作者の制御が間接的である (物語の生成はユーザ側に任されている)．直接制御と間接制御の問題は，現実のプログラミング，とくに並列プログラミングでは大きな問題である．われわれはまだ間接制御について多くを知らないのが現状である．

筆者は，ゲームに限ったとしても，物語プログラミングでは，リアルプログラミングと同様，時間認知と空間認知の問題が取り上げられるべきだと思う．松岡氏らの研究はその意味でも目が離せない．

なお，計算機で複雑さを楽しむといったときには，人工生命 (Artificial Life) について言及しないですませる

わけにはいかない．強引に我田引水すれば，人工生命を楽しむというのは，上で述べた (わわわれがまだ多くを知らない) 間接制御が生ずる意外性を楽しむことである．しかしある意味で，人工生命の研究には「脱物語」的なところがあり，それがどう巡りめぐって物語に回帰してくるかに筆者は大きな感心がある．

4. 複雑性とはなにか

前節まで，「複雑さ」を人間にとってわかりにくいというナイーブな意味でつかってきた．ここでは，複雑さとはなんだろうという根源的な疑問について雑感を述べよう．まとまった話でないのはお許しいただきたい．

Kolmogorov が与えた二進数列 (0, 1 からなる数列) の複雑度の定義は，適当な言語 (計算機) を定めたときに，その数列を生成するプログラム記述の最小量というものである．ここで二進数列を，系の振る舞いと拡大解釈して読み替えても差し支えあるまい．また，記述量で測った複雑さなので，これを記述的複雑度と呼んでもよいだろう．

Kolmogorov 複雑度にはいろいろな応用があるが，よく知られているのは，乱数の定義である．直観的にいうと，乱数とは Kolmogorov 複雑度の大きい数列である．つまり，数列を記述するのに，その数列と同じ程度の長さのプログラムが必要なのが乱数というわけである．平たく言い換えると，「乱数は複雑な数列である」となるのだが，ハテ? と思われる方も多いだろう．

Kolmogorov 複雑度 (記述的複雑度) に対して，アルゴリズム (を実現するプログラム) の走行時間を計算量的複雑度という．これも，敢えて平たくいうと，「時間のかかるアルゴリズムは複雑である」となって，少し奇妙な感じである．なお，Kolmogorov 複雑度と計算量的複雑度はそれぞれ別のものの複雑性を計測しているので，ぶつかりあう概念ではない．どちらも数学として確立した概念で，よく研究されている．

これらに対して，Benett が定義した論理深度 (logical depth) という一種の計算量的複雑度がある．これは二進数列 (系の振る舞い) を記述する最短プログラムが実際に要する計算時間に相当するものである (らしい — 残念ながら，筆者にはまだ理解しきれていない)．たとえば，Fermat 予想や，Riemann 予想など，古今の未解決問題の解を 1-0 にコード化した数列は非常に大きい論理深度をもつ．また，Benett の言葉にしたがえば，DNA の塩基配列はものすごく長い生物学的プロセスの所産であり，論理的に深いとなる．

かなり昔，チューリングマシンの上の短いプログラムでなるべく長く走らせるという競争が行なわれたことがある (ビジービーバー競争と呼ばれた)．筆者は 1976 年ごろ，各種のプログラミング言語の公正な比較を行なうことを考えていて，再帰呼び出しのオーバヘッドだけをうまく計測できるようなベンチマークを発見した．タライ関数と名付けたこの関数は，3 個の整数を引数にする関数で次のように定義される (その後，野崎昭弘氏により，3 個の実数でも停止することが証明された)．

$$
\begin{aligned}
&f(x,y,z) \equiv \\
&\quad \textbf{if } x \le y \textbf{ then } y \\
&\qquad \textbf{else } f(f(x-1,y,z), \\
&\qquad\qquad f(y-1,z,x), \\
&\qquad\qquad f(z-1,x,y))
\end{aligned}
$$

一目でわかるように，再帰呼び出し以外は，1 を引く演算と，条件判定しか行なっていない．ベンチマークテストでは $n = 5$ あるいは $n = 6$ ぐらいにして $f(2n, n, 0)$ を実行させる，これくらい小さな数でも，計算時間はかなり多く，分オーダーの時間がかかる．しかし，タライ関数は次のように定義される非常に簡単な関数 g

$$
\begin{aligned}
&g(x,y,z) \equiv \\
&\quad \textbf{if } x \le y \textbf{ then } y \\
&\qquad \textbf{else if } y \le z \textbf{ then } z \textbf{ else } x
\end{aligned}
$$

と等価であることが証明できるのである！ タライという名前は，引数のタライ回しの結果，結局，引数のどれかを値として返すだけということに由来している (現在では，Tak 関数とか，Takeuchi 関数と呼ばれる)．

この関数の驚くべきところは，($f(2n, n, 0)$ の場合) 非常に長い計算ステップを要するものの，計算に要するメモリー (具体的には関数の実行履歴を管理するスタック領域) が n の定数倍でしかないことと，途中に出てくる値も -1 から $2n$ の間の小さな範囲でしかないことである．

小さな引数で，長大な計算時間を要する関数として最も有名なのが，2 個の自然数 m, n に対して定義される Ackermann 関数 A である．

$A(m,n) \equiv$
  if $m = 0$ **then** $n+1$
    else if $n = 0$ **then** $A(m-1,1)$
      else $A(m-1, A(m,n-1))$

これは原始帰納的でない帰納的関数の一例として知られているもので，計算時間ばかりではなく，値が，どんな原始帰納的関数（直観的にいえば，再帰呼び出しをつかわずに，通常の四則演算，条件判定およびループだけで書けるような関数）でつくれる値よりも大きくなる．これは計算機にとっては，計算時間ではなくメモリー容量の制限で先にパンクしてしまうことを意味する．また，多倍長整数の計算が主になってしまうので，再帰法のベンチマークとしては相応しくない．

タライ関数の計算量的複雑性は，Stanford 大の Knuth 教授を中心とするグループで精力的に研究された．詳細は省くが，$f(n,0,n+1)$ の実行において else 部分が評価される回数を T_n とすると，任意の $\epsilon > 0$ に対して，n を十分大きくすれば，

$$T_n > n^{(1-\epsilon)n}$$

が成立する．すなわち，タライ関数は引数の与え方により，ほぼ $O(n^n)$ の計算時間がかかるということになる．

タライ関数はその実行挙動もなかなか乱れており予測がつきにくい．このため，並列計算機のベンチマークにも適している．従来，再帰的に定義されたフィボナッチ関数 fib

$fib(n) \equiv$
  if $n \leq 1$ **then** 1
    else $fib(n-1) + fib(n-2)$

などがよくつかわれていたのだが，これは十分に解析された関数であり，並列タスクのスケジューリングに予測をつかうことが可能であった．しかし，タライ関数はそれが通用しないので，より現実の応用に近い状況を並列計算機に課すことができるというわけである．

以上，ちょっと長々とタライ関数について紹介したが，これは複雑性の本質について考える一つのヒントを与えていると思う．まず，記述は十分に短い．しかし，挙動は不規則で，かつ計算量がきわめて大きい．ところが，これと等価な関数 g はあきれるほど単純なのである．つまり，タライ関数は「本質的に」なにも複雑なことをしていない（ちなみに，f と g が等価なことは，自動定理証明の格好の題材ともなり，Boyer と Moore がそれを実現した）．

われわれの感ずる複雑性とはなんなのだろうか．f しか知らない人は，f の挙動を見て，とても複雑と感ずるに違いない．もし f を知らず，f の実行履歴を（あたかもポアンカレ写像のような手段で）ところどころしかピックアップできない状況であれば，その感はさらに深まるに違いない．このとき，実はそれは g だったと知ったときの驚きはいかほどのものになるのだろうか．

同じようなことは，有名なライフゲームについてもいえよう．ライフゲームはクロックをたとえば 1/60 秒ぐらいにすると，雲のようにしか見えない．これだけを見た人は，その挙動に非常に奥深いものを感ずるに違いない．しかし，クロックを落とすと，背後のルールが少し見えてくる．そして，あのマジックナンバー 2 と 3 を見い出したときにどのような感激を覚えるだろうか．

いまの二つの例から窺えることは，仕組みのわかっていない系の複雑さは，観測者にとって相対的であらざるを得ないということである．仕組みがわかった瞬間に複雑でなくなったという事例は歴史上枚挙にいとまがないはずである．

しかし今日，人々が話題にする複雑性は，むしろ仕組みのわかったものの挙動の複雑さというところに向けられているように思われる．その典型例がカオスであろう．カオスを起こす方程式系は実際驚くほど単純である．しかし，その挙動は初期値過敏性に代表されるように，予測が事実上不可能である．

プログラムのデバグにおいて非常にやっかいなものに，文法上は通ってしまうタイプミスがある．これは，マシン上ではプログラムの数ビット程度の差となることが多い．このようなプログラムを実行させたとき，すぐタイプミスとわかるものもあるが，そうでないものはとても質が悪い．カオスの初期値過敏性に通ずるような不思議な挙動を誘発してしまうからだ．こういう意味で，カオスの初期値過敏性と，プログラムのビット過敏性にはなにか合い通ずるものがあると筆者は感じている 実際はカオスの方程式系の中のパラメータの値に対する過敏性と合い通じるというのが正確であろうが．

話が脱線するが，複雑なシステムは，それと近いレベルの複雑さをもったシステムとは相性がいいのだが，単純きわまりないものに致命的に弱いことが多い．たとえば，人間という複雑なシステムは，同じく複雑なほかの

人間とはうまくやっていけるし，複雑さは劣るが各種の細菌ともまあうまくやっていける．病原菌もクスリを使えば，病原菌だけを殺すことも可能だ．しかし，青酸カリのような単純なものには致命的である．計算機システムでも，いまのバグの話など，同じようなことがあるのがとても興味深い．

閑話休題．仕組みがわかっているものの複雑さというのは，記述的な複雑さではないだろう．では，計算量的複雑さかというと，筆者には確信がもてない．計算自身は簡単に見えるものがあるからだ (カオスのシミュレーション自身は量的に大きくない)．すると，その複雑さの数学的定式化はなんなのであろう．予測の困難性なのであろうか．しかしこれは単に言葉の言い換えであって，記述的複雑性の範疇になってしまう．

たとえば，π の十進展開の各桁は，π を求める式を知っていれば，それほど難儀せずに予測することができる．予測ということだけについていえば，カオスだって本来の実数計算ができるのであれば，予測は完璧である．もっとも，本来の実数計算などできっこないないのだから，予測はもともと不可能なのだが…．しかし，予測の困難さだけをいうと，乱数が最も複雑な列となってしまう．

カオスやタライ関数のように，元となるものはわかっていて，しかもそれが十分に短い記述ですむのに，その挙動がそれに不釣合いなほどややこしい，という「複雑さ」の直観的定義 (これも，数多い「複雑さ」のうちの，計算論的な側面の一つにすぎないのだろうが) を，うまく定式化できるのだろうか? それとも，ただ複雑だと感じているだけなのだろうか? なにしろ，複雑さの感覚は相対的なものだという経験的事実もある．定式化するとすれば，Benett の論理深度のようなものをここで持ち出したいのだが，筆者には自信がない．ふたたび Benett の言葉を借りると「構造が (論理的に) 深いのは，一見ランダムに見えるが，微妙に冗長な場合である」なのだが….

話は飛ぶが，前節で触れたような「楽しむための複雑性」もまた異なる「複雑性」のように思われる．乱数的な意外性は無論だめで，カオスもレベルを間違えると意外性の感覚をマヒさせてしまうのではないか? $1/f$ といった話は，「楽しむための複雑性」とどうからむのだろうか．また，ひょっとすると「楽しむための複雑性」は人間の創造にのみ許された特権なのだろうか?

どうやら，手が届きそうで，届かないといった領域に複雑なものがあるということらしい．「複雑」とは永遠の逃水のようなものといえる．近くに寄ると，もっと「複雑」なものが向こうに見えるというわけだ．複雑性の研究とは，いわばタマネギの皮むきなのかもしれない．これはまさに人工知能の研究の性格そのものだと妙なところで感心して，小文をしめくくることとしたい．

[文献]

[1] 松岡正剛: われわれはいかに物語性を獲得したか，人工知能学会誌「大規模知識ベース小特集」，March, 1993.

[2] 竹内郁雄: 物語プログラミングの課題，情報処理学会 IM シンポジウム報告集，1992.

[3] M. Li and P. M.B. Vitanyi: Kolmogorov Complexity and its Applications, in Handbook of Theoretical Computer Science (ed. J. van Leeuwen), Elsevier, 1990.

[4] D.E. Knuth: Textbook Examples of Recursion, in Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy, ed, V. Lifschitz), Academic Press, 1991.